Self Modifying Circuitry -
A Platform for Tractable Virtual Circuitry

Adam Donlin,
Department of Computer Science,
University of Edinburgh,
James Clerk Maxwell Building,
Mayfield Road,
Edinburgh EH9 3JZ.
Scotland

Abstract

The readily available performance advantages, gained in early virtual circuitry systems, are being recouped following advances in general purpose processor architectures and have resulted in a questioning of the tractability of applying virtual circuitry in a general software environment. This paper highlights two primary limitations of existing virtual circuitry systems: technical bandwidth limitations, imposed by the use of a shared peripheral bus to interconnect the configurable logic and host processor; and the abstract complications involved in traversing the hardware software divide within the inherently hardware/software co-design environment of a virtual circuitry system. The Flexible URISC is introduced as an array resident minimal processor architecture with the potential to exploit self-reference and self-modification. Performance results of a prototype implementation of the Flexible URISC architecture demonstrate how peripheral bus bandwidth limitations are overcome by the increased bandwidth available to an array resident configuration, communication and computation agent. A discussion of the programming environment of the Flexible URISC is given, and provides the medium for identifying how the Flexible URISC's single instruction – move *– effectively minimises the hardware/software divide.*

1 Introduction

Increases in FPGA density and flexibility have done much to improve the tractability of the concept of virtual circuitry[1], yet exploitation of custom hardware co-processing has remained the elusive attribute of a restricted set of application classes. This is in contrast to the early calls to revolutionise the general computing industry with the widespread deployment of virtual circuitry. General purpose processor systems, as has been revealed by case study[7], have also been successful in recouping a significant amount of the readily available performance advantages gained by early virtual circuitry systems.

Key issues in sustaining the elusiveness of a widespread application of virtual circuitry stem from constraints imposed by the environment in which virtual circuitry is typically deployed, and the complexities of negotiating the hardware software divide. An a-typical setting for a virtual circuitry system is one where an FPGA is coupled to a host processor system via the host peripheral bus. The FPGA works exclusively as a slave co-processor, under the management of software component executing on the host processor. Virtual circuitry systems have been classified into various forms[1] yet all are typically applied in this common setting and, hence, are constrained by its common trappings. These trappings include the technical bandwidth limitations of the host peripheral bus and the penalties incurred in traversing the hardware/software boundary.

The primary aim of this paper is to present an alternative environment for the deployment of virtual circuitry, in which the limitations of the traditional virtual circuitry environment have been addressed. Eliminating these limitations correspondingly increases the tractability of a general deployment of virtual circuitry. The following section provides a brief introduction to the Ultimate RISC architecture. Section 3, presents a discussion of a *Flexible* Ultimate RISC (Flexible URISC), which forms the

[1] For reasons discussed in [2], the term virtual circuitry is used in preference to virtual hardware

primary vehicle for the tractable implementation of virtual circuitry. Section 4 relates the Flexible URISC to existing FPGA based processors and is followed, in section 5, by the exploration of technical requirements and implementation issues. The prospect of self modifying circuitry is introduced in section 6 and it's exploitation to support tractable virtual circuitry is discussed in section 7. Section 8 discusses a programming environment for the exploitation of the Flexible URISC and characterises both runtime and compile-time versions of the system.

2 The Ultimate RISC

The Ultimate RISC(URISC)[6] is a minimal processor architecture with only one instruction: *move memory to memory*. Computation is achieved by migrating devices onto the system bus, then mapping those devices into the memory space of the URISC processor core. For example, the core of the URISC machine possesses no ALU; instead, an ALU component resides on the system bus and its registers are mapped into the memory space of the URISC core. By moving operands to and from the memory addresses corresponding to the registers of ALU components, arithmetic computations may be performed.

The datapath of the URISC core, the Instruction Execution Unit(IEU), is shown in figure 1. The primary responsibility of the core is to implement the move instruction and, since move is the only instruction, no operand decoding is necessary. Unconditional jumps are possible by moving the target address into the memory mapped Program Counter(PC). Conditional jumps are made by adding the contents of a memory mapped ALU condition code register to an branch address that is then written back to the PC. This allows the destination of the jump to be offset by the truth or false value contained in the condition code register.

A standard URISC machine could be implemented using traditional hardware or on a configurable logic array. The lean resource requirements of the URISC control and data paths increase the feasibility of an implementation on configurable logic, yet the static nature of the standard URISC machine excludes any benefits of dynamic reconfiguration. The remainder of this paper, therefore, focuses on the Flexible URISC machine.

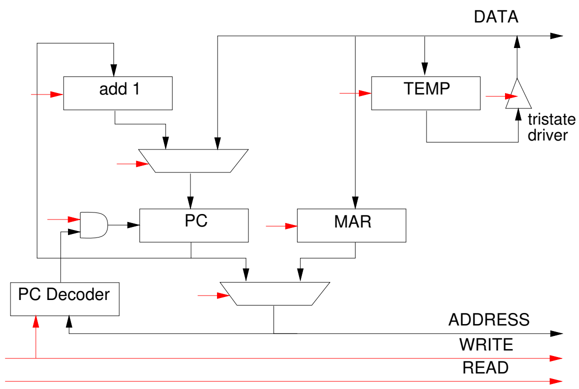

Figure 1: URISC core datapath, the Instruction Execution Unit

3 The Flexible URISC

Figure 2 shows the abstract and physical implementations of a Flexible URISC system on an FPGA. The primary difference between a standard URISC system and a Flexible URISC system is the Flexible URISC's exploitation of dynamic reconfiguration. Specifically, in the Flexible URISC system, dynamic reconfiguration is used to vary the devices which are currently resident on the system bus.

The Flexible URISC core resides on the FPGA, alongside a set of Swappable Logic Units(SLUs)[1]. There is a direct correspondence between the set of SLUs resident on the configurable array and the devices currently resident on the Flexible URISC system bus. Indeed, the input and output registers of each SLU are mapped into the memory space of the Flexible URISC allowing each SLU device to be accessed in the same manner as a piece of static hardware. Using dynamic reconfiguration, SLUs are dynamically placed and replaced, allowing the set of available devices on the URISC system bus to expand and contract as required.

The Flexible URISC exploits a memory-style coprocessor interface, such as the FastMap^{tm} processor interface[3] of the Xilinx XC6200 series FPGA[11]. Notedly, there is no explicit implementation of a system bus utilising the configurable routing resources of the FPGA. The Flexible URISC can, instead, exploit the random access nature of the co-processor interface to move instruction operands to and from the input and output registers of the SLUs resident on the array. Here, the Flexible URISC is acting primarily as a communication agent, transferring operands between SLUs and memory. A more detailed discussion of the

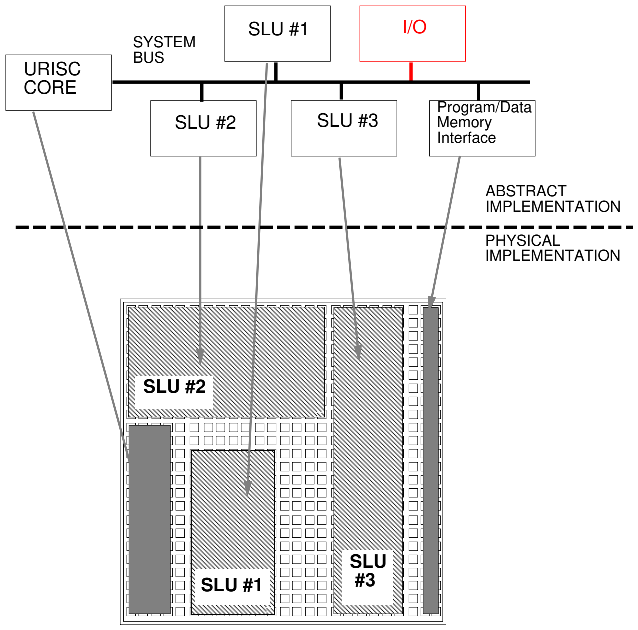

Figure 2: The Flexible URISC Architecture

use of the URISC as a communications agent within the virtual circuitry paradigm has already been undertaken in earlier work[2].

The main focus of this paper, and the following sections is to consider the further benefits gained from the exploitation of the co-processor interface to provide the URISC a role of being a combined communication, computation and configuration agent.

4 FPGA based Processor Systems

A number of previous implementations of a processor core on reconfigurable logic have been developed or suggested[8, 9, 10, 5]. A first distinction between these and the Flexible URISC system is that where the Flexible URISC has been specifically designed to be an entirely stand-alone, self contained processor system, previous systems have relied on the services of an external host processor or auxiliary FPGA. The services typically provided are those to effect dynamic reconfiguration.

DISC[8], an advanced FPGA based processor, exploits partial reconfigurability and facilitates relocatable instruction-grain SLUs. Even so, the services of an auxiliary FPGA are relied upon to provide a partial reconfiguration service that is used in the placement, removal and relocation of instruction SLUs. DISC's Linear Hardware Space, a rigid communications medium, exploits the vertical routing tracks of the DISC FPGA and restricts the relocatability of SLUs to vertical movements only. Instruction SLUs are significantly aware of the DISC environment and require additional internal logic to interface themselves with the overall environment.

The Flexible URISC, due to the lack of an explicit system bus, allows for the placement and relocation on virtually any unoccupied area of the cell array which is of the appropriate dimensions. SLU designs may be of an irregular shape, and are not constrained to the horizontal geometry imposed in the DISC environment. SLUs are more freely designed with a geometry and layout suited to an efficient implementation of the functionality being modelled. Furthermore, there is no additional interfacing logic required within an SLU other than it's input and output registers. The the use of the random access nature of the co-processor interface to implement inter-SLU communication provides any necessary decoding facilities.

Complexity of processor core, and hence the programming models they support, is an additional area where existing systems diverge from the Flexible URISC. The processor cores of the OneChip and NanoProcessor systems, for example, are significantly more complicated than the Flexible URISC core and aim to present a relatively complicated instruction interface.

The fact that the Flexible URISC exploits a move instruction is notably more significant than the fact that only one instruction is supported. Indeed, the primitive move instruction has significant benefits for negotiating the hardware/software boundary, as discussed in section 8.

5 Technical Implementation and Requirements

A prototype of the Flexible URISC core has been implemented using Xilinx XC6200 series Reconfigurable Processing Units (RPUs) and a Xilinx XC6200DS compliant prototyping board. The XC6200DS reference specification defines a system consisting of a single XC6200 series FPGA, dedicated SRAM and PCI Bus interfacing logic. A primary novel contribution of the Flexible URISC system is the presence of an *array resident* configuration agent. By this, it is meant that the Flexible URISC core is capable of dynamically reconfiguring the FPGA device upon which it is currently residing. At the abstract level, this is simply a matter of mapping the configuration memory of the FPGA into the memory map of the Flexible URISC core.

Some key technological advances in the XC6200 RPUs have made the implementation of an array resident configuration agent possible. The ability for user array logic to access the internal control logic of the RPU, such as the configuration address and data busses, is fundamental. The "Open Architecture" of the XC6200 series is an important non-technical feature, allowing essential access to detailed information regarding low level programming interface and bitstream formats. Since the Flexible URISC machine must interface directly with the configuration interface of the RPU, specific details regarding this interface must be attainable.

Additionally, the fact that the FastMap^{tm} interface inherently presents a memory style interface means only a narrow semantic gap need be bridged in mapping the FastMap^{tm} into the Flexible URISC's address space. The amount of interfacing logic required to map the configuration memory into the overall memory map of the URISC core, therefore, is minimised. Alternative configuration interfaces would perhaps require external address decoding or translation, increasing the complexity of the processor core. In total, three conceptually distinct memory interfaces must be mapped into the memory map of the Flexible URISC core. System RAM for holding program and data segments is a basic requirement, derived from a standard URISC machine. In addition to configuration memory, it is also necessary to map the state memory of the user logic resident on the array into the URISC memory space. Mapping of user logic state facilitates the use of the Flexible URISC core as an inter-SLU communications agent.

User state memory is also accessed via the FastMap^{tm} interface and the act of mapping configuration memory also maps state memory into the address space. Extra measures are necessary, however, as state memory cannot be accessed directly, in the same manner as configuration memory. State memory is, instead, addressed by horizontal column. Since a maximum of 32 bits may be accessed and columns potentially contain 128 bits of state data, a series of "map" registers define which bits of the array column are to be masked out. These extra measures could be implemented in hardware, in the Flexible URISC datapath. Instead, they are left as a primarily system-level software task, for reasons discussed in the programming environment section. A software solution also upholds the simplicity and purity of the Flexible URISC core's implementation.

6 Self Modifying Circuitry

By mapping the host FPGA's configuration memory into it's own memory space, the Flexible URISC gains the interesting attribute of self-reference. Indeed, any circuitry that takes advantage of the XC6200 series ability to drive internal array control logic gains the potential for self reference. Circuitry exploiting this self-referentiality to drive the internal FastMap^{tm} control, address and data busses, can be considered "self-modifying".

6.1 The Self Modification Taboo

Traditional software which has access to its program text and data segments possesses an analogous potential for self reference, and hence, self modification. In modern software engineering practices, however, the exploitation of such properties is rare and taboo. For large software systems, this is a justified notion as the unruly application of self modification makes systems particularly difficult to debug. Current generation processor architectures reinforce this taboo through read only instruction caches. Significant cache penalties await programs which override the memory protection facilities of modern operating systems, since modified sections of the text segment in the instruction cache must be flushed.

Efficiency, however, is a primary reason for exploiting self modification. Limited memory, storage, and processing time in early computer systems justified the use of self modification to gain increased code flexibility whilst limiting resource utilisation.

Contemporary virtual circuitry systems find themselves in an analogous situation to early software systems. FPGA device densities, although improving, are still considered limited and configuration penalties remain high. Performance advantages are therefore to be gained from exploiting self modification as the technique for altering the configuration of a resident circuit. The main benefit for self modifying circuitry is the reduced time spent on configuration. The availability of partial reconfigurability is important in allowing the self modifying circuit to remain active whilst part of the datapath is modified. Since older FPGA architectures, which require full reconfiguration of the entire array, render circuitry inactive during full reconfiguration, facilitating self modifying circuitry in these architectures is practically impossible.

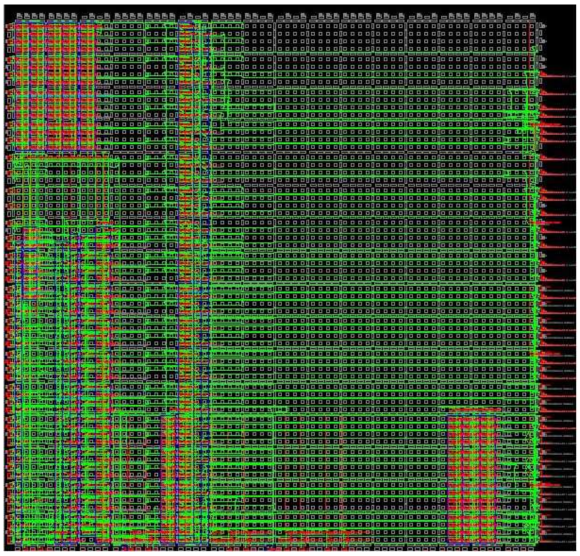

Figure 3: XC6216 implementation of Flexible URISC core

7 Practicable Virtual Circuitry

Increased device level performance in FPGAs has resulted from advances in fabrication technologies, combined with partially reconfigurable architectures. Bottlenecks on the host peripheral bus have become a primary limitation restricting the deployment of virtual circuitry within general software systems. Most FPGA development cards interface an FPGA with a main processor system via the host's PCI bus. Interactions between the FPGA and host must be mediated against other devices resident on the PCI bus.

The self contained nature of the Flexible URISC allows peripheral bus bandwidth limitations to be avoided. As a self-modifying system, the Flexible URISC has direct, unmediated access to the host FPGA's FastMap^{tm} programming interface. Program and circuitry data, stored in on-board SRAM, is directly accessible to the Flexible URISC via a dedicated, on board memory bus. The combination of both these properties within the Flexible URISC core allows for the high bandwidth transfer of program and circuitry across the dedicated memory bus and the fast direct reconfiguration of the host FPGA by self-modification.

System Level Integration(SLI) is a promising approach in combating bandwidth limitation directly. Here, a configurable array and general processor are tightly integrated on the same die. Application of this approach is evident in the RAW architecture which defines a tiled processor architecture and tightly integrated configurable array [4]. Combining a static processor core on the same die as an FPGA represents a potentially more efficient solution to the bandwidth limitation problem.

Since the widespread fabrication and availability of "single die" systems is unlikely within the near future, it should be noted that the Flexible URISC is an equitable present day prototype of such single die solutions. Rather than physically integrating both components on a single die, the Flexible URISC core is tightly integrated with the configurable array by residing on it. Directly interfacing with the host array's configuration interface serves to further tighten the integration. Experiences and performance results gained from the Flexible URISC system should be considered characteristic of those to be gained in the next generation of tightly integrated processor and FPGA systems.

The choice of processor core to be integrated on the same die as an FPGA highlights another attribute of the Flexible URISC system. Integrating a complex processor core on the same die as an FPGA results in a system which has a traditional separation between hardware and software. To exploit the custom computing facilities of the integrated FPGA, it remains necessary to traverse between the notion of software, defined by the instruction set of the integrated processor core, to the notion of hardware, defined the custom circuitry implemented on the FPGA. In this situation, a significant hardware/software divide remains between the integrated processor and configurable array. As will be seen through the discussion of the Flexible URISC programming environment, an important attribute of the URISC system is its ability to narrow the hardware software divide. The benefits of the integration of a Flexible URISC core remain, even following the advent of potentially more efficient solutions to the problem of bandwidth limitation.

7.1 Prototype Performance Analysis

The successful implementation of early prototypes of the Flexible URISC system has allowed for some simple performance analysis. The main aim of this analysis was to assess the degree to which the Flexible URISC can exploit its self contained and self modifying nature in accessing the available memory and configuration interfaces. Of particular interest is the bandwidth available between the Flexible URISC and its program memory. Since all URISC program data and circuitry resides in on board SRAM, the bandwidth available on the dedicated memory interface sets an upper bounds on the level that the array may be self modified.

The Flexible URISC was assigned a simple task, designed to reveal the number of instructions executed. The Flexible URISC was assigned the task of filling a buffer of a known size with a known constant. Starting with the constant c at buffer position 0, a Flexible URISC program was coded which repeatedly moved the constant from position 0 to position n (effectively copying it from position 0 to n). A 16-stage state machine implements the control logic for the prototype Flexible URISC. Every move instruction executed by the Flexible URISC core, therefore, takes 16 clock cycles, four of which involve transactions with the on board memory interface.

Observing the number of buffer cells filled on a given execution of the benchmark URISC program directly corresponds to the number of URISC instructions executed within the period of execution. Since the number of memory transactions incurred per instruction is also defined, executing the benchmark program for a known time period reveals the number of memory transactions completed within that time period. To observe, experimentally, how the memory bandwidth varies with time, the benchmark was repeatedly executed within a range of times, spanning from 0 to 55ms. Execution was performed on a Flexible URISC core operating at a modest 8 MHz, although prototype Flexible URISC cores have also been operational at clock speeds up to 33MHz. The graph shown in figure 4 details a smooth linear progression in the number of instructions executed as the execution period is extended. Using these figures, memory bandwidth exploited by the Flexible URISC core can be calculated. Table 1 details the bandwidth observed experimentally at 8MHz and extrapolated to cores operating at 16 and 33MHz.

These figures are particularly promising in contrast to reported bandwidths attained when negotiating the PCI bus. The maximum reported PCI bandwidth of around 7Mb/s has important consequences on the feasibility of any traditional software based system, such as certain styles of the Sea of Accelerators model of virtual circuitry, attempting to emulate a Flexible URISC machine. Enhancements to the simplistic, unpipelined implementation of the Flexible URISC are conceivable. Of particular interest are approaches to addressing the severe under- utilisation of available memory bandwidth. The prototype, which only issues four memory transactions per 16 clock cycles, would be significantly enhanced by the careful application of pipelining. The benefits of increased utilisation of the memory bus bandwidth must, of course, be weighted against additional array resources consumed by the pipelined implementation.

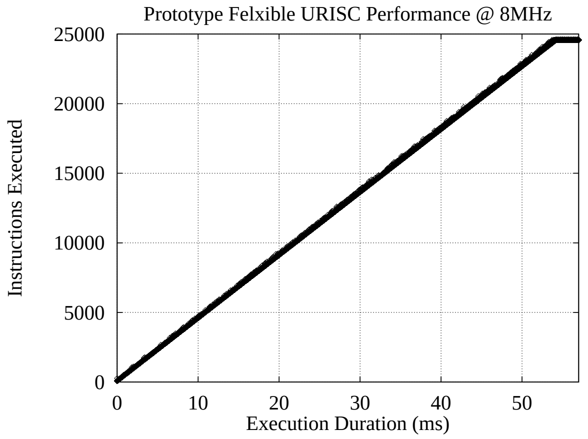

Figure 4: Performance Results of the Prototype Implementation

Clock Speed	Instrs per Sec	Mem Cycles per Sec	Bandwidth (Mb/s)
8 MHz	500,000	2,000,000	8
16 MHz	1,000,000	4,000,000	16
32 MHz	2,000,000	8,000,000	32

Table 1: Flexible URISC Memory Bandwidth

As testimony to its lean implementation, the prototype Flexible URISC, as implemented on a Xilinx XC6216 RPU, occupies 800 function units – approximately 19% of the available function units of a XC6216. The same design implemented on the larger XC6264 RPU consumes only *4%* of the function unit resources. Areas of un-utilised function units are available for the implementation system bus SLUs, provided appropriate routing resources are available. A design heuristic in the course of implementing the Flexible URISC core was to exploit chip-length wires before length 16 routing, length 16 before length 4 and so on. Since system bus SLUs are more likely to rely on local routing before chip length routing, the availability of local routing resources allows un-utilised regions of function units to be more readily exploited.

8 The Flexible URISC Programming Environment

A suitable programming environment must be defined before the novel technical features of the Flexible URISC may be fully exploited. In defining this environment, the overall system context of the Flexible URISC must be considered. Since the primary motivation for the Flexible URISC is to develop a vehicle for the efficient implementation of virtual circuitry, a discussion of the particular model of virtual circuitry supported by Flexible URISC is warranted.

8.1 System Context

The Flexible URISC has been designed as an autonomous, stand-alone system. The programming environment should, however, take into account the relationship between the Flexible URISC and any host processing system. This relationship is clarified as follows.

The Flexible URISC's self-contained and self modifying nature allows it to be an autonomous processing system. The primary system context, an autonomous Flexible URISC, comprises of a central RPU, upon which the Flexible URISC resides, combined with RAM and ROM devices. RAM is primarily used as a program and data store. The bitstream definitions of SLUs used within the program are also held in system RAM. A ROM device provides an appropriate boot configuration of the Flexible URISC. To facilitate a stand alone system, the Flexible URISC is capable of processing immediately at the end of the configuration cycle. Non-volatile RAM is a potentially interesting substitute for a ROM device, upholding the Flexible nature of the Flexible URISC by allowing further self-reference and self modification of the boot sequence.

In the traditional approach to virtual circuitry, a software component executing on the host processor provides runtime virtual circuitry management. Integrating the Flexible URISC and a general purpose processor system effectively results in the creation of a closely coupled multiprocessor system. The metaphor is furthered by exploiting a shared memory interface to facilitate communication between processors. Notably, the Flexible URISC retains entire responsibility for management and implementation of the virtual circuitry model. The complete programming environment could conceptually, by virtue of its autonomous nature, be implemented entirely on the Flexible URISC itself. Integration with a general purpose processor provides the possibility of implementing the programming environment on the general processor. The facilities of the general purpose processor are suitably attuned to implementing the interactive development process.

8.2 The Flexible URISC model of Virtual Circuitry

Two primary models of virtual circuitry, the Sea of Accelerators and the Parallel Harness, have been defined in [1]. Both of these systems can be implemented on the Flexible URISC. Indeed, the definition of Sea of Accelerators gives some consideration to the use of a software agent to transfer data between SLUs. Results from the discussion of practicable virtual circuitry, however, suggest the intractability of this technique, when implemented as an external software system.

Rather than simply implementing one of these models, however, the Flexible URISC provides enough facilities to consider a third model of virtual circuitry – the *Sequential Algorithmic* Model. Contextually, this model rests between the parallel harness and sea of accelerator models. With the Sea of Accelerators model, which exploits no routing, to one side and the Parallel Harness model of hard routing on the other, the Sequential Algorithmic Harness implements efficient *software* routing.

The Sequential Algorithmic model utilises the communications agent abilities of the Flexible URISC to effect processing. The model is sequential as a result of the serialising effect of processing move instructions within the Flexible URISC core. A Flexible URISC program defines, via a series of move instructions, the order of transfer of operands between system bus SLUs. The precise order of system bus transfers can be explicitly described in a lengthy series of moves. More attractive, however, is the potential to exploit existing computational resources residing on the system bus to allow an *algorithmic* definition of complex communication patterns.

8.3 Programming Model

Programming the Flexible URISC in its native instruction set is undesirable. Hand construction of large programs solely with move instructions and absolute addressing as the only addressing mode would be cumbersome and error prone. Given this, a suitable formalism for the expression of a Flexible URISC program must be selected. Some initial consideration is given, however, to the overall development and execution model of a Flexible URISC program.

The Flexible URISC exploits a series of pre-defined SLUs as elements of execution. This is similar to the approach taken in DISC and diverges from the runtime derivation of in-line hardware modules, as suggested for the self-configuring processor[5]. On the fly derivation of hardware modules incurs significant runtime penalties and relies on a native instruction set rich enough to allow traditional software execution of an equivalent software description of the hardware module whilst the hardware module is derived.

The Flexible URISC makes a significant departure from the traditional virtual circuitry execution model. In traditional systems, processing is performed by combining periods of traditional software execution interspersed by periods of hardware custom coprocessing. In contrast, the Flexible URISC makes insists *all* computation is performed in custom hardware modules. Whilst this model was not particularly tractable in previous systems, the technical features of the Flexible URISC used to support the Sequential Algorithmic virtual circuitry model increase the feasibility of hardware execution.

The lack of "on the fly" SLU derivation requires the programming environment include facilities for the construction of the system SLUs. Concern that appropriate SLUS may be undefined when required for the performance of a specific program task is implies a fault on the programmers part. A task of the development process is to ensure the appropriate hardware modules are defined to support execution.

Since the Flexible URISC is, essentially, a dataflow machine a primary requirement of the programming formalism is that it adequately capture dataflow information. The DISC programming environment consists of a rich set of retargetable C tools. In this instance, C is a reasonable programming formalism, reflecting the imperative nature of the DISC core and traditional instruction set interface. Conceptually, C may be used as the Flexible URISC programming formalism, yet this requires additional functionality on the part of the compiler, for the derivation of a suitable dataflow between instruction SLUs to implement the required functionality. An attractive alternative is the use of functional programming languages. Functional languages express both computation and dataflow. Furthermore, the use of higher order functions may provide a suitable formalism for expressing dynamic reconfiguration and self modification.

8.4 Narrowing the Hardware Software Divide

By mapping the configuration and state memory of the host FPGA into its own memory space, the Flexible URISC raises an important opportunity to narrow the hardware software divide. The Flexible URISC move instruction is the primary vehicle by which the hardware software divide is narrowed. The single move instruction effects computation by moving operands to and from system bus SLUs. The same instruction also effects reconfiguration of the host FPGA by moving configuration data from system memory to the configuration memory which has been mapped into the Flexible URISC's memory map.

The point to be noted is that the same instruction is used to both configure and compute. The transition between effecting computation and configuration is highly *transparent* – effectively being an attribute only of the source or destination address of the move instruction. Given a series of Flexible URISC move instructions, the distinction between instructions for computation and instructions for configuration is not immediately apparent.

As well as bridging the gap between computation and configuration, the move instruction narrows the boundary between hardware and software. Considering the software tier of a Flexible URISC program to be the microcode style program defining the movement of operands and the hardware tier to be the processing of operands by system bus circuitry residing on the configurable array. No processing of any interest may occur independently, on any single tier. Realistic processing, instead, involves using basic move to transfer operands between tiers. i.e., transferring operands from system RAM to and from state memory of the host FPGA. The Flexible URISC provides a single instruction which transparently negotiates the hardware/software and computation/configuration boundaries. Effectively, the move instruction is a single interface to hardware, software, and dynamic configuration.

Reducing the hardware software divide is of interest for systems involving hardware software co-design. Systems exploiting virtual circuitry inherently require hardware software co-design issues be addressed. Narrowing the hardware software boundary is of particular benefit in these situations, reducing the focus on interfacing issues and facilitating a transparent transition between hardware and software system components.

8.5 Static and Dynamic environments

Two styles of Flexible URISC environment are envisioned. A compile time system providing a highly efficient implementation of a pre-defined configuration schedule and a dynamic system implementing a primitive operating-system style demand configuration system.

8.5.1 The Static Environment

A compile time system trades performance against flexibility. The costly burden of runtime decision making is avoided by extracting a static reconfiguration schedule, at the cost of overall system flexibility. Increased flexibility makes the dynamic approach more applicable in a general software environment where no static reconfiguration schedule is available.

Loss of flexibility in the static environment is offset by increased opportunities to apply a number of performance enhancing techniques. A novel approach to reconfiguration, for example, is applicable. Traditional virtual circuitry systems employ distinct phases of computation and configuration. Configuration phases are immediately followed by computation phases intended to take maximum advantage of the newly configured circuitry, recouping any reconfiguration penalty. Notedly, processing must halt for a significant period whilst configuration is underway. Utilising the information contained in the static reconfiguration schedule, however, allows the advance determination of configuration deadlines. This, combined with the Flexible URISC's ability to transparently mix configuration and computation instructions at a very fine grain, allows the compiler to begin configuration *in advance* of the execution of dependant computation instructions. Fine grain intermixing of configuration instructions allows a minimum impact on the level of computation instructions being processed. The lack of a static reconfiguration schedule in the dynamic environment restricts the application of such interleaved configuration and computation. Any application of the technique would be analogous to probabilistic page pre-fetching in virtual memory systems.

Further, technology specific techniques, may be exploited in the compile-time environment. As discussed in [2], careful alignment of system bus SLU interfaces and choice of map register values allows, multiple independent moves to be made in a single instruction. Wildcarding may also be exploited to implement multicasting.

8.5.2 The Dynamic Environment

The dynamic environment, in the model of a primitive operating system, implements two levels of programming. A privileged "system program" is charged with the sole responsibility of implementing demand dynamic reconfiguration of the host FPGA. Dynamic environment programs express computation dataflow and contain definitions of the appropriate system bus SLUs to effect the defined computational dataflow. In contrast to static environment programs, which contain integrated configuration, computation and circuitry definitions, no configuration dataflow is defined in user programs. The system program, instead, defines a general configuration dataflow for all user programs.

A dynamic environment compiler supports the use of a "procedural" interface to access the facilities of the system program. Upholding the operating system metaphor, user programs make Flexible URISC "system calls" to request, for example, the placement of a particular system bus SLU. Signalling a return to the traditional model of configuration followed by computation, system calls in the dynamic environment block the caller. Following a model of co-operative multitasking, blocking system calls allow the system program to resume executing, begin the demand paging process or possibly context switch to an alternative user program. A pre-emptive multitasking system is conceivable, but requires modifications to the Flexible URISC core, to introduce timing interrupts.

Additional operating system analogues that may be applied in the dynamic environment include page-faulting and memory protection. Indeed, protection issues are of particular interest to support the enforcement of the distinct privileged and user modes of execution. Examples of privileged resources include the configuration interface which, in the dynamic environment, should only be accessed by the system program. Even more compelling is the need to protect memory mapped non-volatile storage, as introduced in the system context discussion, to avoid corruption of the boot image by unprivileged user programs.

9 Conclusions and Future Work

Literature has shown that traditional performance gains in virtual circuitry systems are waning in the light of significant increases in the raw processing power of general purpose processors. It is the belief of the author, however, that virtual circuitry still has a significant contribution to make to the general field of computing. The advent of "Systems on Silicon" is perceived as a particularly valid and promising new environment allowing virtual circuitry to be successfully integrated with a general software system, without suffering adverse technical bandwidth limitations that have choked existing systems.

The Flexible URISC has been introduced in this paper as a combined computation, communication and configuration agent. By exploiting its ability for self-reference and self-modification, the Flexible URISC represents a practical midpoint between existing virtual circuitry systems and impending future systems exploiting SLI. A basic analysis of the prototype implementation has shown the Flexible URISC has the potential to combat the significant technical limitations of the existing virtual circuitry environment. This result provides some initial evidence of advantages to be gained by systems exploiting SLI.

Future plans for the Flexible URISC involve the development of its programming environment. This facilitates the use of the Flexible URISC as a vehicle for exploring the application hardware/software co-design when, via the Flexible URISC, the boundaries between hardware/software are minimised. Dynamic Protocol Construction, within the field of active networking, is also a planned application area. Here, the aim is to exploit the autonomous nature of the Flexible URISC to develop a stand-alone protocol processing network agent. The Flexible URISC is combined with a memory mapped network interface to provide a framework for supporting the dynamic execution of dynamically defined communication protocols. Specifically, dynamic protocols are defined and implemented as fragments of Flexible URISC programs - essentially a "circlet", the virtual circuitry equivalent of an applet. Utilising the memory mapped network interface of the supporting Flexible URISC system, protocol processing circlets may then be deployed into the active network fabric which is, itself facilitated by the execution of "system" protocol circlets on the same Flexible URISC infrastructure.

References

[1] G. Brebner. A Virtual Hardware Operating System for the Xilinx XC6200. In R. W. Hartenstein and Manfred Glesner, editors, *Proc. 6th International Workshop on Field-Programmable Logic and Applications, FPL'96*, pages 327–336, Darmstadt, Germany, September 1996. Springer-Verlag.

[2] G. Brebner and A. Donlin. Runtime Reconfigurable Routing. In José Rolim, editor, *Parallel and Distributed Processing*, volume 1388 of *LNCS*, pages 25–30. Springer-Verlag, 1998.

[3] S. Churcher, T. Kean, and B. Wilkie. The XC6200 FastMap^{TM} processor interface. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications : 5th international workshop*, volume 975 of *LNCS*, pages 36–43, Oxford, United Kingdom, August/September 1995. Springer-Verlag.

[4] E. Waingold *et al.* Baring it all to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, September 1997.

[5] P. C. French and R. W. Taylor. A self-reconfiguring processor. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 50–59, Napa, CA, April 1993.

[6] D. W. Jones. The Ultimate RISC. *Computer Architecture News*, 16(3):48–55, June 1988.

[7] M. Shand. A Case Study of Algorithm Implementation in Reconfigurable Hardware and Software. In *Proc. 7th International Workshop on Field Programmable Logic and Applications*, volume 1304 of *LNCS*, pages 333–343. Springer, 1997.

[8] M. J. Wirthlin and B. L. Hutchings. A dynamic instruction set computer. In P. Athanas and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 99–107, Napa, CA, April 1995.

[9] M. J. Wirthlin, B. L. Hutchings, and K. L. Gilson. The Nano Processor: A low resource reconfigurable processor. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 23–30, Napa, CA, April 1994.

[10] R. D. Wittig. OneChip: An FPGA Processor with Reconfigurable Logic. Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, 1995.

[11] Xilinx. *The Programmable Logic Data Book*. Xilinx Inc, San Jose CA, 1996.